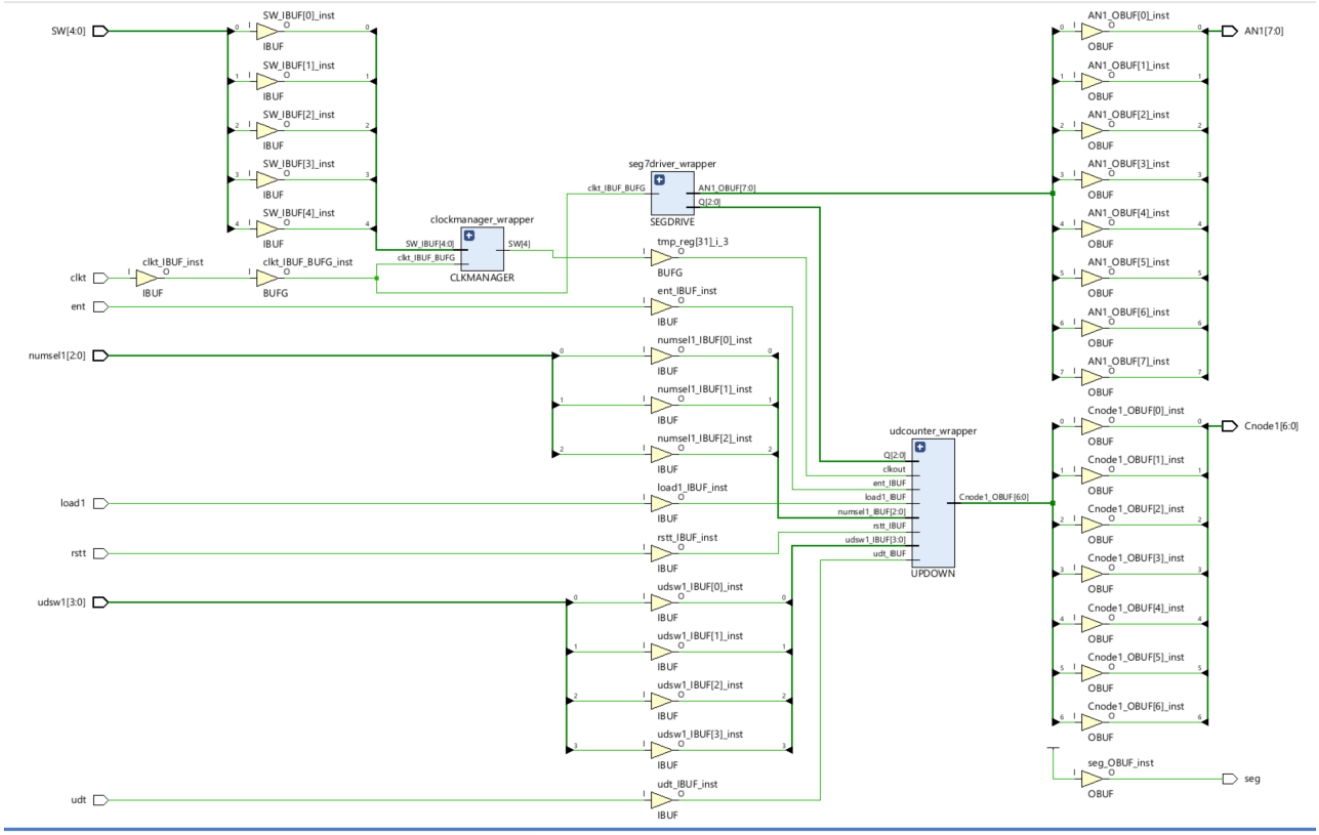
ECE3300L Lab6 Group H Report (Sherwin Sathish & Mohamed Hamida)

SCHEMATIC():



Xdc for top.v:

```
## This file is a general .xdc for the Nexys A7-100T
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clkt}];# IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clkt}];


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { SW[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }]; #IO_L3N_T0_DQS_EMCCLK_14
Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }]; #IO_L6N_T0_D08_VREF_14
Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { udt }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { rstt }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { ent }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { load1}]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { udsw1[0] }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports { udsw1[1] }]; #IO_L15P_T2_DQS_RDWR_B_14
Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { udsw1[2] }]; #IO_L23P_T3_A03_D19_14
Sch=sw[11]
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { udsw1[3] }]; #IO_L24P_T3_35 Sch=sw[12]
```

```
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { numsel1[0] }]; #IO_L20P_T3_A08_D24_14
Sch=sw[13]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { numsel1[1] }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { numsel1[2] }]; #IO_L21P_T3_DQS_14
Sch=sw[15]

## LEDs
#set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { sumbit[0] }]; #IO_L18P_T2_A24_15
Sch=led[0]
#set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { sumbit[1] }]; #IO_L24P_T3_RS1_15
Sch=led[1]
#set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { sumbit[2] }]; #IO_L17N_T2_A25_15
Sch=led[2]
#set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { sumbit[3] }]; #IO_L8P_T1_D11_14
Sch=led[3]
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { coutbit }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14
Sch=led[5]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14
Sch=led[6]
#set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14
Sch=led[7]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { LED[8] }]; #IO_L16N_T2_A15_D31_14
Sch=led[8]
#set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14
Sch=led[9]
#set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { LED[10] }]; #IO_L22P_T3_A05_D21_14
Sch=led[10]
#set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { LED[11] }];
#IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { LED[12] }]; #IO_L16P_T2_CSI_B_14
Sch=led[12]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14
Sch=led[13]
#set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14
Sch=led[14]
#set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14
Sch=led[15]

## RGB LEDs
#set_property -dict { PACKAGE_PIN R12   IOSTANDARD LVCMOS33 } [get_ports { LED16_B }]; #IO_L5P_T0_D06_14
Sch=led16_b
#set_property -dict { PACKAGE_PIN M16   IOSTANDARD LVCMOS33 } [get_ports { LED16_G }]; #IO_L10P_T1_D14_14
Sch=led16_g
#set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 } [get_ports { LED16_R }]; #IO_L11P_T1_SRCC_14
Sch=led16_r
#set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { LED17_B }]; #IO_L15N_T2_DQS_ADV_B_15
Sch=led17_b
#set_property -dict { PACKAGE_PIN R11   IOSTANDARD LVCMOS33 } [get_ports { LED17_G }]; #IO_0_14 Sch=led17_g
#set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { LED17_R }]; #IO_L11N_T1_SRCC_14
Sch=led17_r

##7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[6] }]; #IO_L24N_T3_A00_D16_14
Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[5] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[4] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[2] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[1] }]; #IO_L19P_T3_A10_D26_14
Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { Cnode1[0] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { seg }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { AN1[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { AN1[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { AN1[2] }]; #IO_L24P_T3_A01_D17_14
Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { AN1[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { AN1[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { AN1[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { AN1[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { AN1[7] }]; #IO_L23N_T3_A02_D18_14
Sch=an[7]

##Buttons
```

```
#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { rst }]; #IO_L3P_T0_DQS_AD1P_15
Sch=cpu_resetn
#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports {  load1 }]; #IO_L9P_T1_DQS_14 Sch=btnc
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { dir }]; #IO_L4N_T0_D05_14 Sch=btnu
#set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { BTNL }]; #IO_L12P_T1_MRCC_14 Sch=btnl
#set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { BTNR }]; #IO_L10N_T1_D15_14 Sch=btnr
#set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { BTND }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

Code Detail:

`timescale 1ns / 1ps

`timescale 1ns / 1ps

```
module CLKMANAGER
(
   input clk,
   input rst,
   input [4:0] SW,
   output reg clkout =0
   );
   reg [31:0] sel;



   always@(posedge clk or posedge rst)
   begin: DREG
      if(rst)
         sel<= 32'd0;
      else
         sel<=sel+1;
   end

 always@(SW)
 begin


   case(SW)
         5'd0: clkout <= sel[0];
         5'd1: clkout <= sel[1];
         5'd2: clkout <= sel[2];
         5'd3: clkout <= sel[3];
         5'd4: clkout <= sel[4];
         5'd5: clkout <= sel[5];
         5'd6: clkout <= sel[6];
         5'd7: clkout <= sel[7];
         5'd8: clkout <= sel[8];
         5'd9: clkout <= sel[9];
         5'd10:clkout <= sel[10];
         5'd11:clkout <= sel[11];
         5'd12:clkout <= sel[12];
         5'd13:clkout <= sel[13];
         5'd14:clkout <= sel[14];
         5'd15:clkout <= sel[15];
         5'd16:clkout <= sel[16];
         5'd17:clkout <= sel[17];
         5'd18:clkout <= sel[18];
         5'd19:clkout <= sel[19];
         5'd20:clkout <= sel[20];
         5'd21:clkout <= sel[21];
```

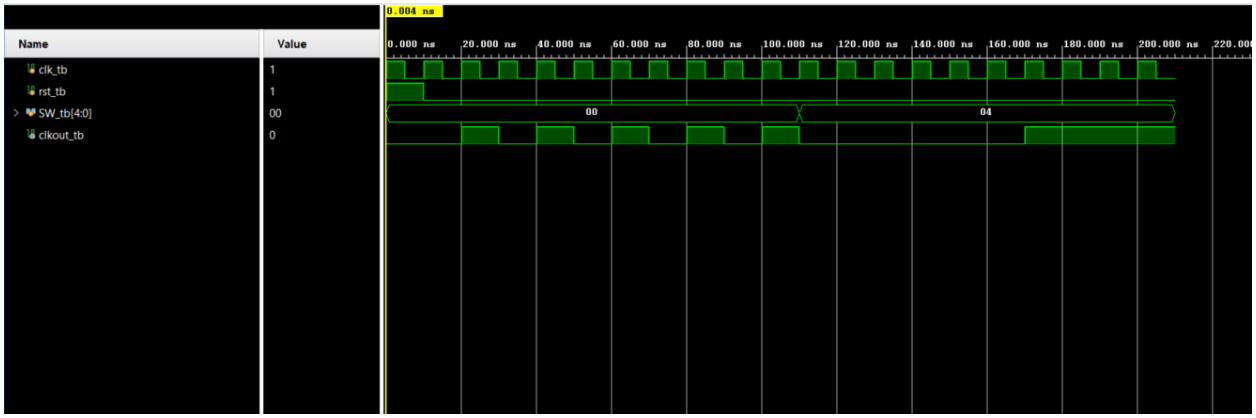
```
        5'd22:clkout <= sel[22];
        5'd23:clkout <= sel[23];
        5'd24:clkout <= sel[24];
        5'd25:clkout <= sel[25];
        5'd26:clkout <= sel[26];
        5'd27:clkout <= sel[27];
        5'd28:clkout <= sel[28];
        5'd29:clkout <= sel[29];
        5'd30:clkout <= sel[30];
        5'd31:clkout <= sel[31];

    endcase
  end


endmodule
```

The purpose of the clock manager is to have a counter and a 5-bit mux to output a clk with various levels of frequency depending on the integer div in this case. First we set up the counter by using an always loop that accounts for the reset, and sets up the different frequencies for all 32 values. Then we created an always loop that takes effect when the switches are changed, and we used a case statement for all 32 values of the 5-bit switches. Depending on the clkout that is selected by the switches, the frequency could vary for each clkout. We used the non-blocking assignment <= so that the code doesn't have to be done in order from top to bottom.

```
`timescale 1ns / 1ps


module CLKMANAGER_TB (

  );

  reg clk_tb,rst_tb;
  reg [4:0] SW_tb;
  wire clkout_tb;


  CLKMANAGER COMP(
  .clk(clk_tb),
  .rst(rst_tb),
  .SW(SW_tb),
  .clkout(clkout_tb)
  );

  initial
    begin
    clk_tb = 1;
    end

  always
    begin
    #5 clk_tb = ~clk_tb;
    //#20 SW_tb = SW_tb+1;
    end
  initial
```

```
    begin
      rst_tb = 1;
      SW_tb <= 0;
     #10
      rst_tb = 0;
     #100
      SW_tb = 4;
     #100
     $finish;
    end

endmodule
```

The purpose of this testbench is to test the code in the design source. We instantiate the variables of the design source code by calling that class,so that we can test them in this simulation source. We set the initial values for the inputs, and we use an always loop to make sure that the clock is toggling. We set a value for the switch to analyze the change in frequency.
EXAMPLE FOR CLKMANAGER:




```
`timescale 1ns / 1ps


module UPDOWN(
    input clk1,
    input rst1,
    input en,
    input ud,
    input load,
    input [3:0] udsw,
    input [2:0] numsel,
    output reg [31:0] tmp
    );

    always@(posedge clk1 or posedge rst1)
    begin
    if(en)

      begin
        if(rst1)
            begin
                if(ud)// 0 is up,  1 is down
                tmp<= 32'd99999999;
                else
                tmp<= 32'd0;
```

```
         end

    else
        begin
            if(ud)
            begin
                if(tmp==32'd0)
                    tmp <= 32'd99999999;
                else
                    tmp <= tmp-1;
                    end
      else if(load)
begin
    if(numsel==3'b000)
        begin
        tmp[31:4]=tmp[31:4];
        tmp[3:0]= udsw;
        end
        else if(numsel==3'b001)
        begin
        tmp[31:8]=tmp[31:8];
        tmp[3:0]=tmp[3:0];
        tmp[7:4] = udsw;
        end
        else if(numsel==3'b010)
        begin
       tmp[31:12]=tmp[31:12];
       tmp[7:0]=tmp[7:0];
       tmp[15:12]= udsw;
        end
        else if(numsel==3'b011)
        begin
        tmp[31:16]=tmp[31:16];
        tmp[11:0]=tmp[11:0];
        tmp[15:12]= udsw;
        end
        else if(numsel==3'b100)
        begin
        tmp[31:20]=tmp[31:20];
        tmp[15:0]=tmp[15:0];
        tmp[19:16]= udsw;
        end
        else if(numsel==3'b101)
        begin
        tmp[31:24]=tmp[31:24];
        tmp[19:0]=tmp[19:0];
        tmp[23:20]= udsw;
        end
        else if(numsel==3'b110)
        begin
        tmp[31:28]=tmp[31:28];
        tmp[23:0]=tmp[23:0];
        tmp[27:24]= udsw;
        end
        else
        begin
        tmp[28:0]=tmp[28:0];
        tmp[31:28]=udsw;
```

```
            end
    end


            else
                    begin
                    if(tmp==32'd99999999)
                    tmp<=0;
                    else
                    tmp<=tmp+1;


                    end


             end
           //tmp = tmp;
       end
       else
       tmp=tmp;
    end

endmodule
```

The purpose of this code is to count up or down and also load in values. The always@(posedge or rst) was used because this is a sequential circuit , and the non-blocking assignment <= is used as well because this code runs in parallel. Except when loading the blocking assignment = is used because the counter has to pause so that we can load in values. The if statement with en is used first because without the en the updown counter wouldn't run. The code either counts up or down and sends the result to a binary to bcd converter, so when we load values it's in normal binary, but it then gets converted to bcd. The numsel is used to signal which digit needs to be changed for the load and the udsw is what value would the user like to input.

```
`timescale 1ns / 1ps

module UPDOWN_TB(

   );
   reg clk1_tb,rst_tb;
   reg en_tb;
   reg ud_tb;
   reg l;
   reg [3:0] u;
   reg [2:0] n;

   wire [31:0] cnt_tb;

   UPDOWN COMP (
   .clk1(clk1_tb),
   .rst1(rst_tb),
   .en(en_tb),
   .ud(ud_tb),
```

```
  .load(l),
  .udsw(u),
  .numsel(n),
  .tmp(cnt_tb)
  );

  initial
     begin
     clk1_tb = 0;
     rst_tb = 1;
     ud_tb = 0;
     en_tb = 1;
     #10
     rst_tb = 0;
     l=1;
     n=1;
     u=4;
     #1000
     $finish;
     end

  always
     begin
     #5 clk1_tb = ~clk1_tb;
     end

/* always
     begin
     #20 rst_tb = ~rst_tb;
     end*/
     endmodule
```

The purpose of this testbench is to test the code in the design source. We instantiate the variables of the design source code by calling that class,so that we can test them in this simulation source. We set the initial values for the inputs, then we use an always loop to make sure that the clock is toggling, and I had an always loop for reset just to test it out.

```
module bin2bcd(
  input [31:0] bin,
  output reg [31:0] bcd
  );

integer i;

always @(bin) begin
   bcd=0;
   for (i=0;i<32;i=i+1) begin                                    //Iterate once for each bit in input
number
   if (bcd[3:0] >= 5) bcd[3:0] = bcd[3:0] + 3;             //If any BCD digit is >= 5, add three
          if (bcd[7:4] >= 5) bcd[7:4] = bcd[7:4] + 3;
          if (bcd[11:8] >= 5) bcd[11:8] = bcd[11:8] + 3;
```

```
            if (bcd[15:12] >= 5) bcd[15:12] = bcd[15:12] + 3;
            if (bcd[19:16] >= 5) bcd[19:16] = bcd[19:16] + 3;
            if (bcd[23:20] >= 5) bcd[23:20] = bcd[23:20] + 3;
            if (bcd[27:23] >= 5) bcd[27:23] = bcd[27:23] + 3;
            if (bcd[31:27] >= 5) bcd[31:27] = bcd[31:27] + 3;
            bcd = {bcd[31:0],bin[12-i]};                    //Shift one bit, and shift in proper bit
from input
    end
end
endmodule
`timescale 1ns / 1ps
module bin2bcd_tb(
    );
    reg [31:0]bin_tb;
    wire [31:0] bcd_tb;

    bin2bcd COMP (
    .bin(bin_tb),
    .bcd(bcd_tb)
    );
    initial
        begin
        bin_tb = 45;
        //$finish
        end


endmodule
```

The purpose of this code is to implement the double dabble algorithm. The “double dabble” algorithm is commonly used to convert a binary number to BCD. The binary number is left-shifted once for each of its bits, with bits shifted out of the MSB of the binary number and into the LSB of the accumulating BCD number. After every shift, all BCD digits are examined, and 3 is added to any BCD digit that is currently 5 or greater. This works because every left shift multiplies all BCD digits by two. Since BCD digits cannot exceed nine, a pre-shift number of five or more would result in a post-shift number of ten or more, which cannot be represented. Adding three to any BCD digit greater than five does two things: first, at the next shift, the 3 that was added becomes 6, and that accounts for the difference in binary and BCD codes

```
`timescale 1ns / 1ps

module segdisplaydriver(
    input [31:0] inDigit,
    output reg [6:0] Cnode,
```

```
output dp,
output reg [7:0] AN,
input nexysCLK, // 100MHz
output reg divided_clk = 0 // 10kHz => 10ms period, 0.5ms ON, 0.5ms OFF
);
reg [3:0] singledigit = 0;
reg [3:0] refreshcounter = 0;
// Calculate division value = 100MHz / (2 * desired frequency) - 1 => 10kHz => 4999

localparam div_value = 25000;

integer counter_value = 0;

always @(posedge nexysCLK)
begin
   if (counter_value == div_value)  // For every (div_value) clock cycles, reset counter back to 0
      counter_value <= 0; // Use <= for parallel & same time, = for sequential - one after the other
   else
      counter_value <= counter_value + 1;
end

// divide clock
always @(posedge nexysCLK)
begin
   if (counter_value == div_value)
      divided_clk <= ~divided_clk; // Flip signal
   else
      divided_clk <= divided_clk; // Keep signal the same
end
/*CLOCK DIVIDER CODE*/

always @(posedge divided_clk)
begin
   refreshcounter <= refreshcounter + 1;
end

always @(refreshcounter)
   begin
      case(refreshcounter)
         4'b0000: singledigit = inDigit[3:0];    // digit 1 value (right digit)
         4'b0001: singledigit = inDigit[7:4];    // digit 2 value
         4'b0010: singledigit = inDigit[11:8];    // digit 3 value
         4'b0011: singledigit = inDigit[15:12];    // digit 4 value
         4'b0101: singledigit = inDigit[19:16];    // digit 5 value missing 4'b0010
         4'b0110: singledigit = inDigit[23:20];    // digit 6 value
         4'b0111: singledigit = inDigit[27:24];    // digit 7 value
         4'b1000: singledigit = inDigit[31:28];    // digit 8 value (left digit)
      endcase
   end

always @(refreshcounter)
begin
   case(refreshcounter)
      4'b0000: AN = 8'b11111110;     // digit 1 ON (right digit)
```

```
            4'b0001: AN = 8'b11111101;     // digit 2 ON
            4'b0010: AN = 8'b11111011;     // digit 3 ON
            4'b0011: AN = 8'b11110111;     // digit 4 ON
            4'b0100: AN = 8'b11101111;     // digit 5 ON
            4'b0101: AN = 8'b11011111;     // digit 6 ON
            4'b0110: AN = 8'b10111111;     // digit 7 ON
            4'b0111: AN = 8'b01111111;     // digit 8 ON (left digit)
            default: AN = 8'bZZZZZZZZ;
        endcase
    end

    assign dp = 1'b1;
    always@(singledigit)
        begin
            case (singledigit)
                4'd0: Cnode<= 7'b0000001; //0
                4'd1: Cnode<= 7'b1001111; //1
                4'd2: Cnode<= 7'b0010010; //2
                4'd3: Cnode<= 7'b0000110; //3
                4'd4: Cnode<= 7'b1001100; //4
                4'd5: Cnode<= 7'b0100100; //5
                4'd6: Cnode<= 7'b0100000; //6
                4'd7: Cnode<= 7'b0001111; //7
                4'd8: Cnode<= 7'b0000000; //8
                4'd9: Cnode<= 7'b0000100; //9
                4'd10:Cnode<= 7'b0001000; //A
                4'd11:Cnode<= 7'b1100000; //B
                4'd12:Cnode<= 7'b0110001; //C
                4'd13:Cnode<= 7'b1000010; //D
                4'd14:Cnode<= 7'b0110000; //E
                4'd15:Cnode<= 7'b0111000; //F
                default: Cnode = 7'b0000000; //DEFAULT CASE EVERYTHING ON
            endcase
        end
endmodule

`timescale 1ns / 1ps

module SEGDRIVE_TB(

    );
    reg [31:0] tmp_SW_tb;
    wire [6:0] Cnode_tb;
    wire dp_tb;
    wire [7:0] AN_tb;


    SEGDRIVE COMP (
    .tmp_SW(tmp_SW_tb),
    .Cnode(Cnode_tb),
    .dp(dp_tb),
    .AN(AN_tb)

    );
```

```
    initial
        begin
        //dp_tb = 1;
        tmp_SW_tb=0;
        #10
        tmp_SW_tb=4;
        #1000
        $finish;
        end




endmodule
```

This code takes an input of 32 switches and depending on when the switches change value, one of the many possible outputs will be selected to display on the 7 segment display. AN and dp are assigned in the beginning to make sure it is only one digit without the decimal place being shown. The different cases range from 0 to 9, and the 32-bit switch value selects the specific digit that needs to be modified. In this particular code 0 represents on, while 1 represents off. There is also a clock divider to make sure that the digits can change asynchronously so that the official updown 32-bit counter will work.

```
        `timescale 1ns / 1ps

module top(
    input clkt,
    input rstt,
    input ent,
    input udt,
    input [4:0] SW,
input load1,
    input [3:0] udsw1,
    input [2:0] numsel1,
    output seg,
    output [6:0] Cnode1,
    output [7:0] AN1
    );
    wire slowclk_out;
    wire [31:0] inDigit;
    CLKMANAGER clockmanager_wrapper(
        .clk(clkt),
        .SW(SW),
        .clkout(slowclk_out)
    );


    wire [31:0] tmp_cnt;
```

```
    UPDOWN udcounter_wrapper(
        .rst1(rstt),
        .en(ent),
        .clk1(slowclk_out),
       .load(load1),
        .udsw(udsw1),
        .numsel(numsel1),
        .tmp(tmp_cnt),
        .ud(udt)
    );

   bin2bcd bin2bcd_wrapper(
        .bin(tmp_cnt),

        .bcd(inDigit)
    );


    SEGDRIVE seg7driver_wrapper(
        .nexysCLK(clkt),
        .inDigit(inDigit),
         .Cnode(Cnode1),
        .dp(seg),
        .AN(AN1)
    );
endmodule


`timescale 1ns / 1ps


module top_tb(

    );
    reg clktq;
    reg rsttq;

    reg [4:0] SW1q;
    reg udtq;
    reg l;
    reg [3:0] u;
    reg [2:0] n;
    wire segq;
    wire [6:0] Cnode1q;
    wire [7:0] AN1q;

   top COMP(
    .clkt(clktq),
    .rstt(rsttq),
    .load1(l),
    .udsw1(u),
    .numsel1(n),

    .SW(SW1q),
```

```
    .udt(udtq),
    .seg(segq),
    .Cnode1(Cnode1q),
    .AN1(AN1q )


   );

    initial
        begin
        clktq = 1;
        udtq =0;
        end

    always
        begin
        #5 clktq = ~clktq;
        //#20 SW_tb = SW_tb+1;
        end
    initial
        begin
            rsttq = 1;

           #10
            rsttq = 0;
           #100
            SW1q = 0;
           #100
        l=1;
        n=0;
        u=0;
        #1000

           $finish;
        end


endmodule
```

The top file basically instantiates the clock, switches, and clock out of the CLKMANAGER. It instantiates the reset, enable, clock, updown, and 32-bit output of the BCD converter from the UPDOWN, and the SEGDRIVE switches, Cnode, dp, and Anode. Two wires are created to connect everything together. The slowclk_out wire connects the output clk of the clk manager to the input of the updown counter clk. The wire tmp_cnt connects the 4-bit output of the updown to the input of the switches, so the values of 0 to 9 can be displayed on the FPGA board.

This code represents the following schematic of a clkmanager that generates 32 random frequencies and then sends it out as the clock of the updown which counts up and down at different frequencies. The 32-bit output represents the binary value, which will be connected to a

converter to convert the value into bcd, which will be connected to the switches of the 7-segment display, so that it could display the BCD on the 7-segment display.

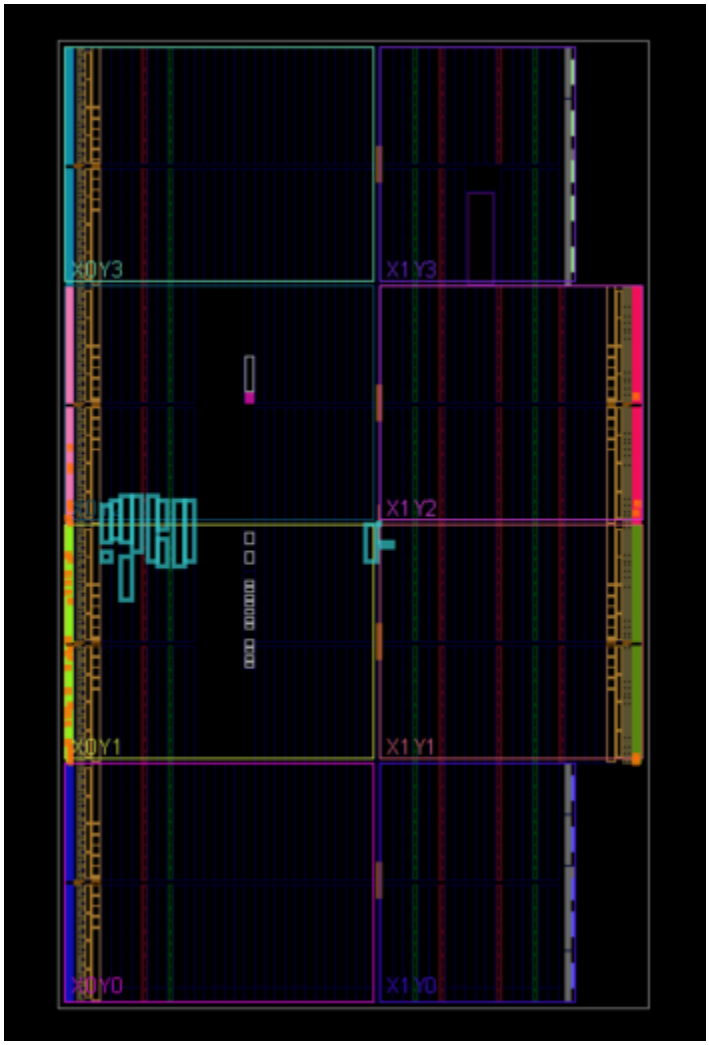
Corner Cases/Error :

One corner case can be the CLKMANAGER Testbench. At first the testbench wasn't working because of :

```
always@(SW)
 begin
```

The problem is that the loop can only run if the SW changes, so we had to change SW to posedge clk or posedge rst, so that the code could effectively work in the testbench.

IMPLEMENTED DESIGN/ TIMING SUMMARY:



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.162 ns	Worst Hold Slack (WHS):	0.320 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	96	Total Number of Endpoints:	96	Total Number of Endpoints:	66

All user specified timing constraints are met.

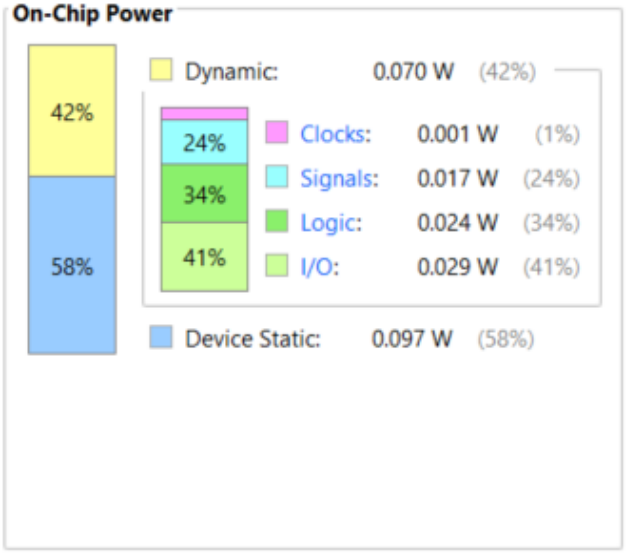
POWER SUMMARY:

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.168 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.8°C**
Thermal Margin:	59.2°C (12.9 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



UTILIZATION:

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	Slice (15850)	LUT as Logic (63400)	Bonded IOB (210)	BUFGCTRL (32)
N top	470	100	12	172	470	33	2
clockmanager_wrapper (CLKMANAGER)	10	32	4	13	10	0	0
seg7driver_wrapper (SEGDRIVE)	16	36	0	18	16	0	0
udcounter_wrapper (UPDOWN)	444	32	8	141	444	0	0

RESOURCE USAGE:

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP
synth_1	constrs_1	Synthesis Out-of-date									472	100	0	0	0
impl_1	constrs_1	Implementation Out-of-date	4.162	0.000	0.320	0.000		0.000	0.168	0	470	100	0	0	0